



Week 10

GUI Tables



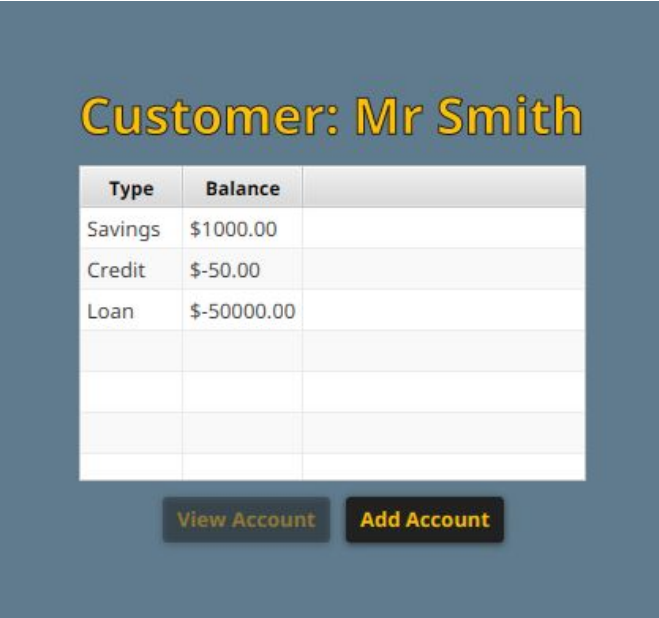
This week

- TableView
- Change Listeners
- Catching Exceptions
- Throwing Exceptions

TableView

TableView<X>

- A TableView<X> displays a list of items of type X.
- A TableView has:
 - A row for each item
 - A column for each property of each item
- e.g. A TableView<Account> has:
 - A row for each Account
 - A column for each property of each Account



Customer: Mr Smith

Type	Balance	
Savings	\$1000.00	
Credit	\$-50.00	
Loan	\$-50000.00	

[View Account](#) [Add Account](#)

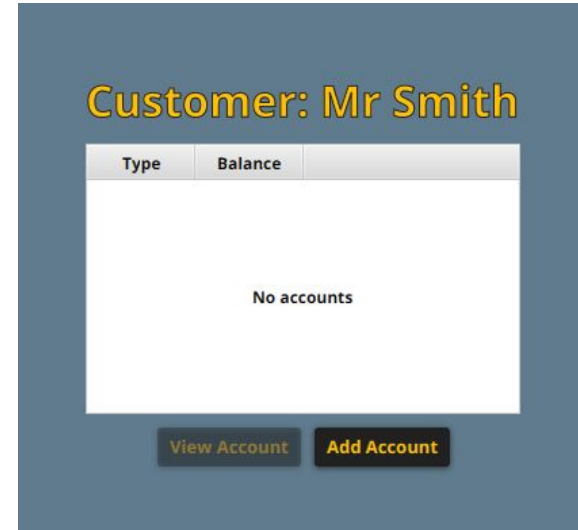
FXML and Java code

- Creating a TableView in FXML:

```
<TableView fx:id="accountsTv" prefWidth="300" prefHeight="200">  
    <placeholder><Label text="No accounts"/></placeholder>  
    <columns>  
        <TableColumn text="Type"/>  
        <TableColumn text="Balance"/>  
    </columns>  
</TableView>
```

- Declaring the TableView in your controller:

```
@FXML private TableView<Account> accountsTv;
```



Linking the TableView to the model

- Two ways to link the view and model

- In FXML:

- ```
<TableView fx:id="accountsTv" items=" ${controller.customer.accounts} ">
```

- In Java:

- ```
accountsTv.setItems ( getCustomer () .getAccounts () );
```

- You must:

- Expose a "customer" property in the controller
 - Expose an "accounts" property in the customer model

Linking each TableColumn to a model property

- Use a `PropertyValueFactory` to link the column to a property value:

```
<?import javafx.scene.control.cell.*?>
```

```
...
```

```
<columns>
```

```
    <TableColumn text="Type">
```

```
        <PropertyValueFactory property="type"/>
```

```
    </TableColumn>
```

```
    <TableColumn text="Balance">
```

```
        <PropertyValueFactory property="balance"/>
```

```
    </TableColumn>
```

```
</columns>
```

- You must expose the following properties in the account model:
 - `type`
 - `balance`

The result...

Customer: Mr Smith

Type	Balance	
Savings	1000.0	
Credit	-50.0	
Loan	-50000.0	

[View Account](#)

[Add Account](#)

Cell value factories

- A cell value factory is generates the contents of a cell. Two options:
 - `PropertyValueFactory` is a cell value factory that just displays a property value.
 - Define your own custom cell value factory to display data how you want.

- A cell value factory is set on the column:

```
TableColumn<Account,String> column = ...;  
column.setCellValueFactory (...);
```

- `TableColumn<X,Y>` has two type parameters:
 - `X` is the type of the item being displayed in the row
 - `Y` is the content type of the cell in this column

Setting a custom cell value factory

- Assign an id to the column:

```
<TableColumn fx:id="balanceClm" text="Balance"/>
```

- In your controller:

```
@FXML private TableColumn<Account, String> balanceClm;  
@FXML private void initialize() {  
    balanceClm.setCellValueFactory(cellData ->  
        cellData.getValue().balanceProperty().asString("$%.2f"));  
}
```

- `TableColumn<Account, String>` means the item for this row is an `Account`, and the cell contents to be displayed is a `String`.

The result...

Customer: Mr Smith

Type	Balance	
Savings	\$1000.00	
Credit	\$-50.00	
Loan	\$-50000.00	

[View Account](#)

[Add Account](#)

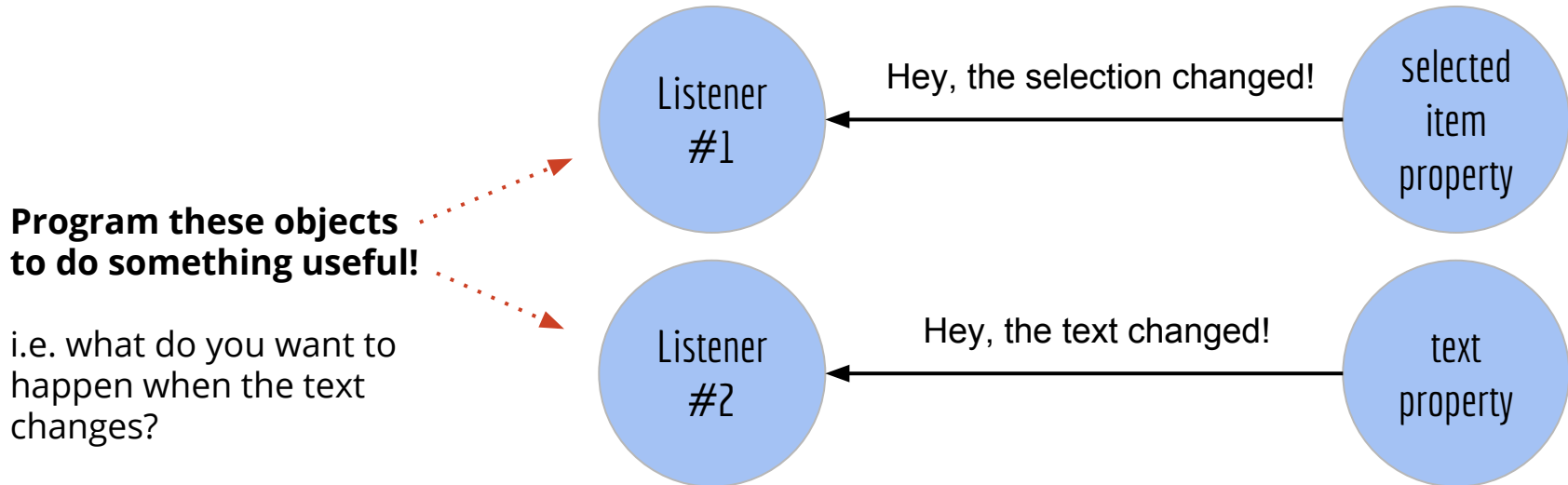


Change Listeners



Listening for changes

- You can listen for changes to anything that is “observable”.
- Whenever the observable changes, the observer/listener is notified.



Change Listeners

- A change listener is any object that implements the `ChangeListener<X>` interface, where `<X>` is the type of value being observed.

- The interface is imported:

```
import javafx.beans.value.*;

public interface ChangeListener<X> {
    void changed(ObservableValue<? extends X> observable,
                X oldValue, X newValue);
}
```

- Register your observer with:

```
observable.addListener(observer);
```

Goal #1: Enable button when account is selected

Customer: Mr Smith

Type	Balance	
Savings	\$1000.00	
Credit	\$-50.00	
Loan	\$-50000.00	

[View Account](#) [Add Account](#)

Customer: Mr Smith

Type	Balance	
Savings	\$1000.00	
Credit	\$-50.00	
Loan	\$-50000.00	

[View Account](#) [Add Account](#)

Solution

- Update the `disable` property of the button whenever the `selectedItem` property changes.

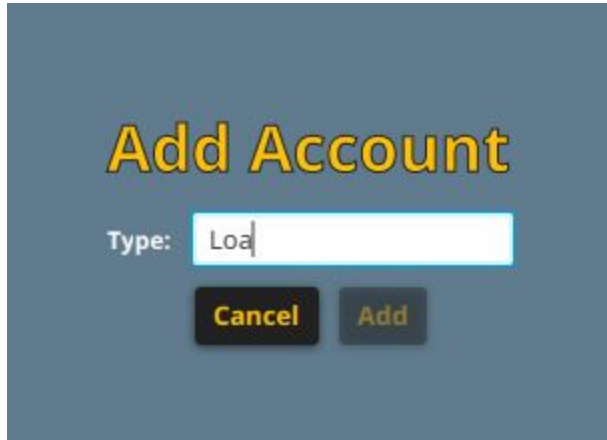
- FXML:

```
<TableView fx:id="accountsTv" items="{controller.customer.accounts}"/>
<Button fx:id="viewBtn" text="View Account"  disable="true"
        onAction="#handleViewAccount"/>
```

- Controller:

```
@FXML private void initialize() {
    accountsTv.getSelectionModel().selectedItemProperty().addListener(
        (o, oldAcct, newAcct) -> viewBtn.setDisable(getAccount() == null));
}
```


Goal #2: Enable button when Type \geq 4 characters

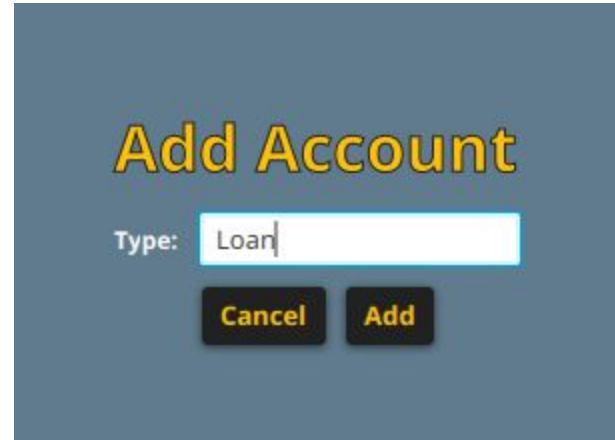


Add Account

Type:

Cancel **Add**

The image shows a form titled "Add Account" on a blue background. Below the title is a label "Type:" followed by a text input field containing the text "Loa". At the bottom of the form are two buttons: "Cancel" and "Add". The "Add" button is currently disabled (grayed out).



Add Account

Type:

Cancel **Add**

The image shows the same "Add Account" form, but the text input field now contains the text "Loan". The "Add" button is now enabled (has a dark background and yellow text).

Solution

- Update the button's `disable` property when the `text` property changes.

- FXML:

```
<TextField fx:id="typeTf"/>
<Button fx:id="addBtn" text="View Account"  disable="true"
        onAction="#handleViewAccount"/>
```

- Controller:

```
@FXML private void initialize() {
    typeTf.textProperty().addListener((o, oldText, newText) ->
        addBtn.setDisable(getAccount() == null));
}
```

Any property can be observed for changes

- Print the account balance whenever it changes:

```
account.balanceProperty().addListener((obs, oldBal, newBal) ->  
    System.out.println("Balance changed from "+oldBal+" to "+newBal));
```

- Print the text of a TextField whenever it changes:

```
nameTf.textProperty().addListener((obj, oldText, newText) ->  
    System.out.println("Text updated to " + newText));
```

- Print the selected toggle whenever it changes:

```
genderTg.selectedToggleProperty().addListener((o, old, now) ->  
    System.out.println("Selected gender: " + now));
```



Catching Exceptions



Exceptions

- Sometimes a method can fail to do its job. In such situations, that method *throws* an “exception”.
- To handle this error, the caller *catches* the exception.
- To know what types of exception a method might throw, refer to the Java API documentation:

<https://docs.oracle.com/javase/8/docs/api/>

API documentation for exceptions

Scanner

```
public Scanner(File source)  
    throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

source - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

API documentation for exceptions

parseInt

```
public static int parseInt(String s)  
    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value or an ASCII plus sign '+' ('\u002B') to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

s - a String containing the int representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

NumberFormatException - if the string does not contain a parsable integer.

Exceptions

- Consider the following program:

```
Scanner scanner = new Scanner(new File("data.txt"));  
int a = Integer.parseInt(scanner.nextLine());  
int b = Integer.parseInt(scanner.nextLine());  
int c = a / b;  
System.out.println(a + " / " + b + " = " + c);  
scanner.close();
```

- What could go wrong?

Exceptions

- Consider the following program:

```
Scanner scanner = new Scanner(new File("data.txt"));
int a = Integer.parseInt(scanner.nextLine());
int b = Integer.parseInt(scanner.nextLine());
int c = a / b;
System.out.println(a + " / " + b + " = " + c);
scanner.close();
```

What if the file is not found?

What if the file is empty?

What if the data is not an integer?

What if b is zero?

- What could go wrong?

A LOT!!!

Traditional error handling: “if” statements

```
if (!new File("data.txt").exists()) {  
    System.out.println("File not found: data.txt");  
    return;  
}  
  
Scanner scanner = new Scanner(new File("data.txt"));  
if (!scanner.hasNextLine()) {  
    System.out.println("No line found");  
    return;  
}  
  
String aStr = scanner.nextLine();  
if (!aStr.matches("[0-9]+")) {  
    System.out.println("Incorrect format for string: " + aStr);  
    return;  
}  
  
int a = Integer.parseInt(aStr);    ... CONTINUED NEXT SLIDE ...
```

Traditional error handling: “if” statements

```
... CONTINUED ...  
String bStr = scanner.nextLine();  
if (!bStr.matches("[0-9]+")) {  
    System.out.println("Incorrect format for string: " + bStr);  
    return;  
}  
int b = Integer.parseInt(aStr);  
if (b == 0) {  
    System.out.println("Divide by zero");  
    return;  
}  
System.out.println(a + " / " + b + " = " + c);  
scanner.close();
```

- Phew!

Handling exceptions with a try-catch block

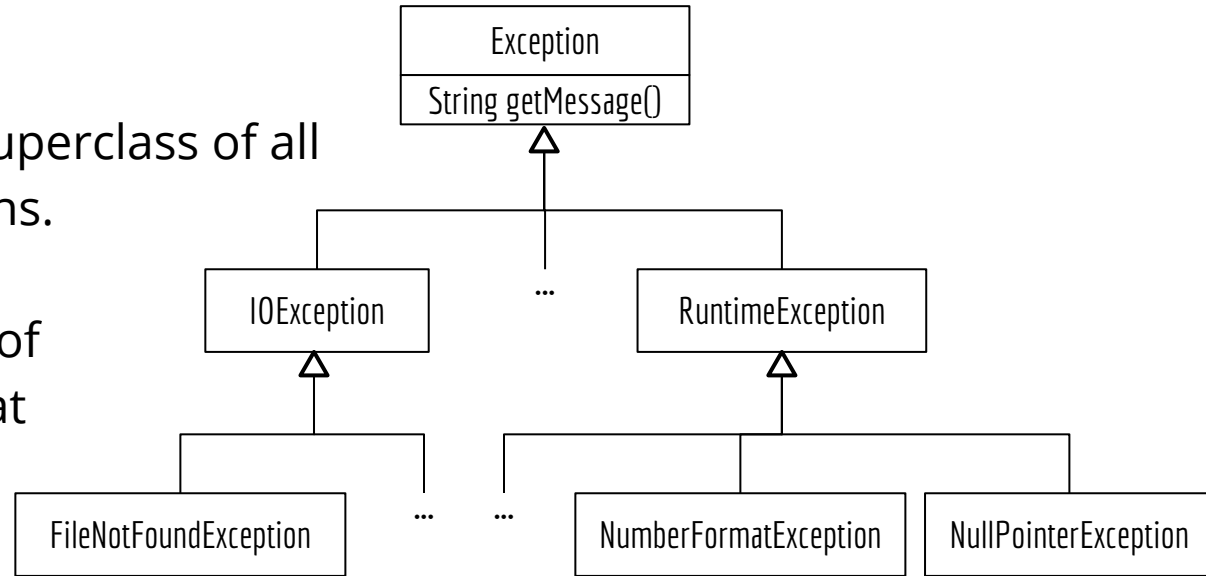
```
try {  
    Scanner scanner = new Scanner(new File("data.txt"));  
    int a = Integer.parseInt(scanner.nextLine());  
    int b = Integer.parseInt(scanner.nextLine());  
    int c = a / b;  
    System.out.println(a + " / " + b + " = " + c);  
    scanner.close();  
} catch (FileNotFoundException e) {  
    System.out.println("File not found: data.txt");  
} catch (NoSuchElementException e) {  
    System.out.println("Not enough lines: data.txt");  
} catch (NumberFormatException e) {  
    System.out.println("Incorrect number format: " + e.getMessage());  
} catch (ArithmeticException e) {  
    System.out.println("Cannot divide by zero");  
}
```

Try to execute this block.

If any line fails due to an exception, stop trying and jump to the catch block for that exception.

Exception Inheritance Hierarchy

- Class `Exception` is the superclass of all exceptions.
- `IOException` is the superclass of all input/output exceptions.
- Catching a superclass of exceptions catches that entire category of exceptions



Catch all

- To catch all exceptions with a single catch block, catch `Exception`:

```
try {
    Scanner scanner = new Scanner(new File("data.txt"));
    int a = Integer.parseInt(scanner.nextLine());
    int b = Integer.parseInt(scanner.nextLine());
    int c = a / b;
    System.out.println(a + " / " + b + " = " + c);
    scanner.close();
} catch (Exception e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

The “finally” block

- A “finally” block always executes.
- e.g. Always close the file after reading, even if an exception occurs:

```
Scanner scanner = null;
try {
    scanner = new Scanner(new File("data.txt"));
    int a = Integer.parseInt(scanner.nextLine());
    int b = Integer.parseInt(scanner.nextLine());
    int c = a / b;
    System.out.println(a + " / " + b + " = " + c);
} catch (Exception e) {
    System.out.println("An error occurred: " + e.getMessage());
} finally {
    if (scanner != null) scanner.close();
}
```

Try-with-resource (Java 7)

- A try-with-resource statement declares a resource that is auto-closed:

```
try (Scanner scanner = new Scanner(new File("data.txt"))) {  
    int a = Integer.parseInt(scanner.nextLine());  
    int b = Integer.parseInt(scanner.nextLine());  
    int c = a / b;  
    System.out.println(a + " / " + b + " = " + c);  
} catch (Exception e) {  
    System.out.println("An error occurred: " + e.getMessage());  
}
```




Throwing Exceptions



Throwing an exception

- If you write a method that can fail, consider declaring that method to throw an exception.
- Examples:
 - `public void withdraw(double amount) throws InsufficientFundsException`
 - `public void addAccount(String type) throws DuplicateAccountException`
 - `public void removeAccount(String type) throws NoSuchAccountException`
- You may define your own exception classes or use a generic exception.

Throwing a generic exception

- The withdraw method throws an exception if the amount is too high:

```
public void withdraw(double amount) throws Exception {  
    if (amount > balance.get())  
        throw new Exception("Insufficient funds");  
    balance.set(balance.get() - amount);  
}
```

- The method header specifies a comma-separated list of exceptions it can throw.

```
public void foo() throws IOException, NumberFormatException
```

Throwing a custom exception

- Define a custom exception as a subclass of Exception:

```
public class InsufficientFundsException extends Exception {  
    public InvalidAmountException() {  
        super("Insufficient funds");  
    }  
}
```

- Use the custom exception:

```
public void withdraw(double amount) throws InsufficientFundsException {  
    if (amount > balance.get())  
        throw new InsufficientFundsException();  
    balance.set(balance.get() - amount);  
}
```

Goal: Show an error if withdraw fails

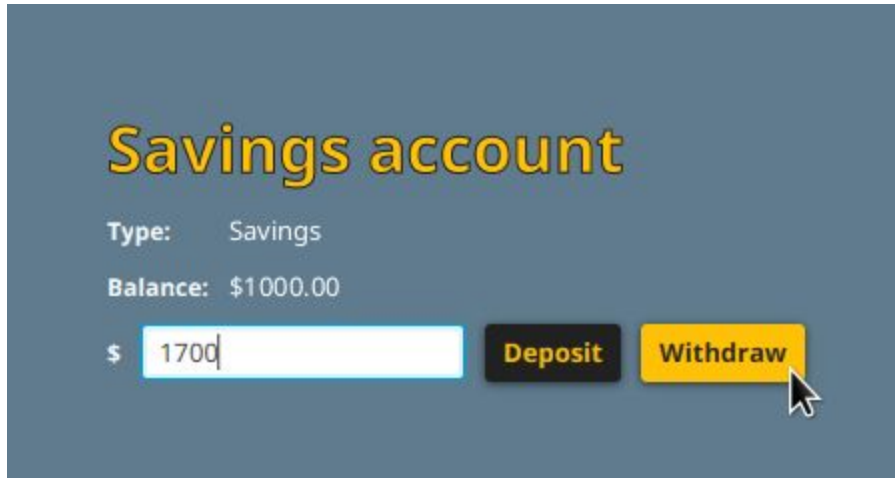
Savings account

Type: Savings

Balance: \$1000.00

\$

Deposit **Withdraw**

A screenshot of a web application interface for a savings account. The background is a solid blue-grey color. At the top, the text "Savings account" is displayed in a bold, yellow, sans-serif font. Below this, the text "Type: Savings" is shown in a smaller, white, sans-serif font. Further down, the text "Balance: \$1000.00" is displayed in the same white font. Below the balance, there is a white input field with a blue border, containing the number "1700". To the left of the input field is a white dollar sign "\$". To the right of the input field are two buttons: a dark grey button with the text "Deposit" in yellow, and a yellow button with the text "Withdraw" in dark grey. A white mouse cursor is pointing at the "Withdraw" button.

An error occurred

Insufficient funds

OK

A screenshot of an error message dialog box. The background is a solid blue-grey color. At the top, the text "An error occurred" is displayed in a bold, yellow, sans-serif font. Below this, the text "Insufficient funds" is shown in a smaller, white, sans-serif font. At the bottom center, there is a dark grey button with the text "OK" in yellow.

The catch or specify requirement

- If you write code that might throw an exception, you must either specify that the exception might be thrown, or catch that exception.

- Option #1: Specifying the exception:

@FXML

```
public void handleDeposit(ActionEvent event)
    throws NumberFormatException, InsufficientFundsException {
    account.deposit(Double.parseDouble(amountTf.getText()));
    setAmount(0);
}
```

- If you don't catch the exception, you push back the catch or specify requirement to the caller.

Uncaught exceptions

- If an exception is thrown all the way to the top level without being caught, the user sees a stack trace:

```
Caused by: java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:497)
    at sun.reflect.misc.Trampoline.invoke(MethodUtil.java:71)
    at sun.reflect.GeneratedMethodAccessor1.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:497)
    at sun.reflect.misc.MethodUtil.invoke(MethodUtil.java:275)
    at javafx.fxml.FXMLLoader$MethodHandler.invoke(FXMLLoader.java:1765)
    ... 50 more
```

```
Caused by: model.InsufficientFundsException: Insufficient funds
    at model.Account.withdraw(Account.java:20)
    at controller.AccountController.handleWithdraw(AccountController.java:35)
    ... 60 more
```

The catch or specify requirement

- Option #2: Catching the exception:

```
@FXML public void handleDeposit(ActionEvent event) {  
    try {  
        account.deposit(Double.parseDouble(amountTf.getText()));  
    } catch (Exception e) {  
        ViewLoader.showStage(e, "/view/error.fxml", "Error", new Stage());  
    } finally {  
        setAmount(0);  
    }  
}
```

- General practice: throw an exception up as high as possible, but catch and handle it before the user sees the stack trace.

Unchecked exceptions

Unchecked exceptions are not subject to the catch or specify requirement. There are two kinds:

- Any subclass of **RuntimeException** is unchecked. Runtime exceptions are typically due programming bugs. Examples:
 - NullPointerException
 - NumberFormatException
 - ArrayIndexOutOfBoundsException
- Any subclass of **Error** is unchecked. Errors are failures of the environment. Examples:
 - OutOfMemoryError
 - IOError